



ALMA MATER STUDIORUM  
UNIVERSITÀ DI BOLOGNA

## **(Laboratorio di) Amministrazione di sistemi**

# **Filtri**

**Marco Prandini**

Dipartimento di Informatica – Scienza e Ingegneria

# Convenzioni

- Il font `courier` è usato per mostrare ciò che accade sul sistema; i colori rappresentano diversi elementi:
  - `rosso per comandi da impartire o nomi di file`
  - `blu per l'output dei comandi`
  - `verde per l'input (incluse righe nei file di configurazione)`
- Altri colori possono essere usati in modo meno formale per evidenziare parti da distinguere nei comandi o indicazioni importanti nel testo
- I parametri formali sono normalmente scritti in maiuscolo e riportati nello stesso colore nel testo che ne descrive l'utilizzo

# Filtri

- Il meccanismo di ridirezione è utilizzato da un set di comandi pensati esattamente per elaborare stream di testo ricevuti via stdin, che producono risultati su stdout, i *filtri*
- Vedremo alcuni dei più comuni
- Molti di questi permettono comunque di operare direttamente anche su file specificati come parametro, da cui prelevano i dati da elaborare in alternativa alla lettura da stdin



# Concatenazione di file – cat / tac

- Il più semplice dei filtri: invocato senza parametri copia stdin su stdout
- invocato con uno o più file come parametri, ne produce in sequenza il contenuto su stdout

```
cat file1 file2
```

qualche minima opzione utile: numerare le righe, evidenziare tab e fine linea, ecc.

```
man cat
```

- **tac** riproduce le righe in ingresso (stdin o file) su stdout in ordine inverso, dall'ultima alla prima



# Impaginazione – less

- Non è realmente un filtro perché l'output è destinato al terminale, ma è tra i comandi più utili per l'uso interattivo
- Posto al termine di una pipeline, intercetta l'output e lo mostra riempiendo lo spazio disponibile sul terminale, permettendone la navigazione per mezzo di vari comandi (tipicamente la pressione di un singolo tasto)
- Comandi principali (ce ne sono altre decine – **man less**):
  - **h** help (dei comandi disponibili)
  - **frecce/pag-su/giu** movimento
  - **F** Follow; scorre fino al termine dell'input e resta in attesa che compaiano nuove righe da mostrare
  - **<N>g** si porta alla riga numero **<N>** (default: 1)
  - **G** si porta al termine del file
  - **/<pattern>** cerca la riga successiva al cursore contenente **<pattern>**
  - **?<pattern>** cerca la riga precedente il cursore contenente **<pattern>**
  - **n** ripete la ricerca fatta in precedenza
  - **N** ripete la ricerca fatta in precedenza, ma nel verso opposto
  - **q** esce da less

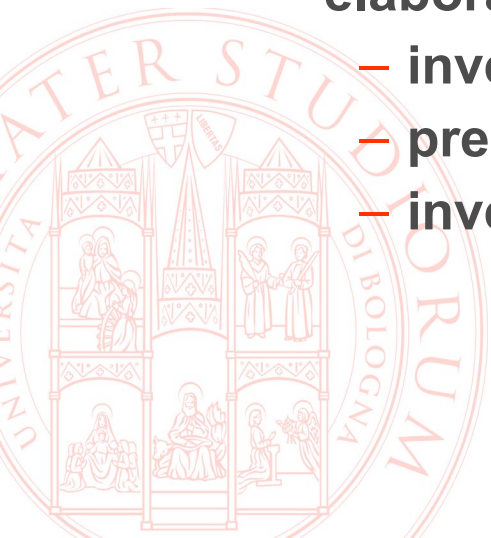
# Il comando rev

- rev è un filtro che permette di invertire l'ordine dei caratteri di ogni linea dello stream in input verso lo stream di output.
- L'utilità del comando è solitamente quella di accompagnare cut nella estrazione di campi la cui posizione sia nota relativamente al fine linea:

```
cat /etc/passwd | rev | cut -f1 -d: -s | rev
```

elabora il contenuto del file /etc/passwd nel seguente modo:

- inverte ogni linea
- prende il primo campo → l'ultimo campo dell'originale
- inverte ogni linea (ripristina il campo selezionato)



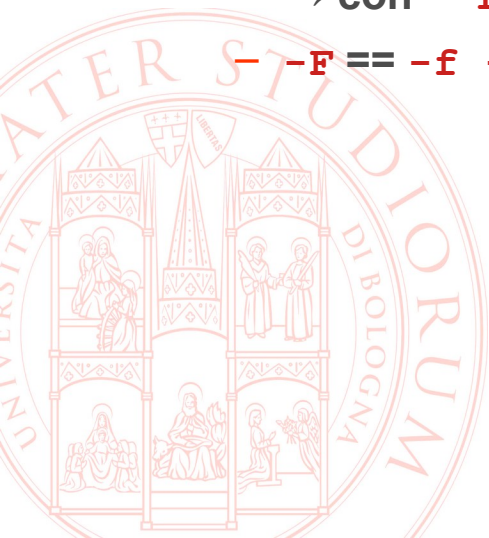
# I comandi head e tail

- **head** è un filtro che permette di estrarre la parte iniziale di un file
  - default: prime 10 righe
  - **-c NUM** produce i primi **NUM** caratteri
    - usando **-NUM** produce tutto il file eccetto gli ultimi **NUM** caratteri
  - **-n NUM** produce le prime **NUM** righe
    - usando **-NUM** produce tutto il file eccetto le ultime **NUM** righe
  
- **tail** è un filtro che permette di estrarre la parte finale di un file
  - default: ultime 10 righe
  - **-c NUM** produce gli ultimi **NUM** caratteri
    - usando **+NUM** produce tutto il file a partire dal carattere **NUM**
  - **-n NUM** produce le ultime **NUM** righe
    - usando **+NUM** produce tutto il file a partire dalla riga **NUM**



# Opzioni particolari di tail

- Un'opzione particolarmente importante di tail è **-f** con cui, dopo aver mostrato le ultime righe di un file, lo si mantiene aperto e si visualizzano in tempo reale eventuali nuove righe che vi vengano appese da altri processi
- **tail -f** quindi può essere usato per monitorare l'output che un processo sta scrivendo su di un file
  - per far questo ignora il raggiungimento di EOF... come termina?
    - con **-f** si può usare **--pid=PID** per fare in modo che alla terminazione del processo PID termini anche tail
  - e se il produttore è in ritardo, e non genera il file in tempo per l'avvio di tail?
    - con **--retry**, tail continuerà a provare finché non riuscirà ad aprire il file
  - **-F == -f --retry**





# Estrazione dei caratteri di una riga – cut

- Il comando cut permette di tagliare parti di righe.
- Il modo di operazione più semplice è attivato dall'opzione **-c**

```
cut -cELENCO_POSIZIONI_CARATTERI [input_file]
```

“ritaglia” le righe, producendo per ogni riga in ingresso una riga in uscita composta dai soli caratteri elencati.

Esempi:

```
cut -c15
```

restituisce solo il 15° carattere

```
cut -c8-30
```

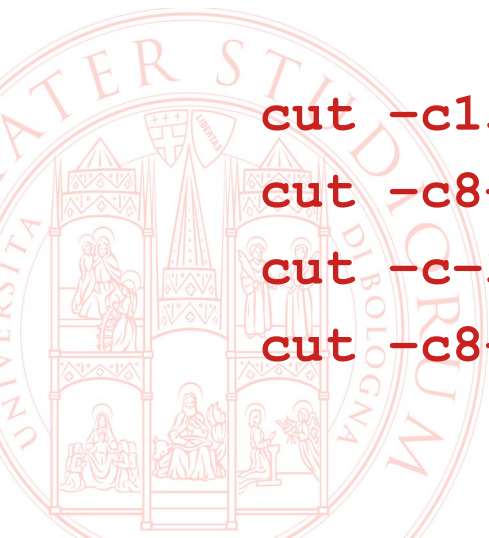
restituisce i caratteri dall'8° al 30°

```
cut -c-30
```

restituisce i caratteri fino al trentesimo

```
cut -c8-
```

restituisce i caratteri dall'ottavo in poi



# Estrazione dei campi – cut

- Su file organizzati a 'record' (uno per riga) per i quali ogni record rappresenta una lista di 'campi' opportunamente delimitati, le opzioni **-d** e **-f** di cut permettono di estrarre uno o più campi di ciascun record

**cut -dCARATTERE\_DELIMITATORE -fELENCO\_CAMPI**

Es. se ci interessa estrarre solo il campo username dal file passwd:

**cat /etc/passwd | cut -f1 -d: -s**

-s evita che vengano prodotte in output le righe che non contengono il delimitatore (che altrimenti sono riprodotte per intero)

Es. se nel campo note metto 'Nome Cognome' e voglio l'iniziale dei cognomi degli utenti:

■ **cat /etc/passwd | cut -f5 -d: -s | cut -f2 -d' ' | cut -c1**

# I comandi sort e uniq

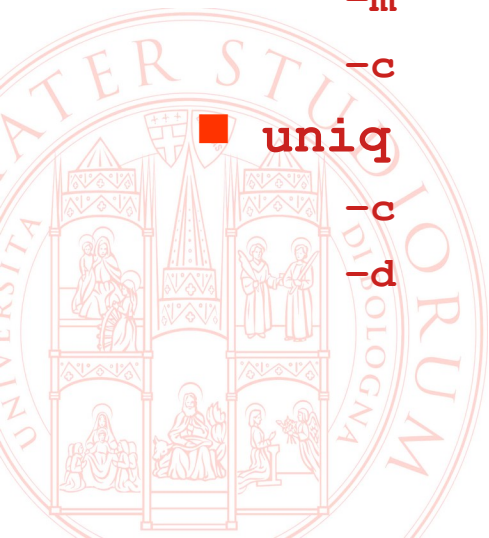
- Per ordinare le linee di uno stream o per individuare le righe duplicate o uniche, unix mette a disposizione i filtri **sort** e **uniq**.

- **sort** ordina in modo lessicale

- ordine dei caratteri =
  - se **LC\_ALL=C** → valore dei byte che li codificano
  - diversamente → ordine stabilito dal *locale* scelto
- opzioni che controllano il comportamento globale
  - u** elimina le entry multiple (equivale a **sort | uniq**)
  - r** reverse (ordinamento decrescente)
  - R** random (permutazione casuale delle righe)
  - m** merge di file già ordinato
  - c** controlla se il file è già ordinato

- **uniq** elimina i duplicati consecutivi

- c** indica anche il numero di righe compattate in una
- d** mostra solo le entry non singole



# sort - opzioni avanzate di ordinamento

- Oltre all'ordinamento di default, sort è in grado di confrontare le righe sulla base di altri criteri

- b ignora gli spazi a inizio riga
- d considera solo i caratteri alfanumerici e gli spazi
- f ignora la differenza minuscole / maiuscole
- n interpreta le stringhe di numeri per il valore numerico
- h interpreta i numeri "leggibili" come 2K, 1G, ecc.

- inoltre, può essere istruito a cercare le chiavi di ordinamento in posizioni specifiche della riga, invece che considerarla per intero

- tSEP imposta SEP come separatore tra campi (default: spazi)
- kKEY chiave di ordinamento – se usato più volte, ordina per la prima chiave, a parità di questa per la seconda, ...

KEY è nella forma (semplificata) **F[.C][,F[.C]][OPTS]**

- **F** = numero di campo
- **C** = posizione (in caratteri) nel campo
- **OPTS** = una delle opzioni di ordinamento **[bdfgiMhnRrV]**

- Es: **sort -t. -k 1,1n -k 2,2n -k 3,3n -k 4,4n**  
ordina un elenco di IP address (byte1.byte2.byte3.byte4)

# Il comando **wc**

- **wc** (word count) è un filtro di conteggio
  - c** conta i caratteri
  - l** conta le linee
  - w** conta le parole (stringhe separate da spazi)



# Ricerca di parti in un testo: Grep

- Esamina le righe del testo in ingresso (su standard input o specificato come elenco di file sulla riga di comando)
- Riproduce in uscita quelle che contengono un pattern corrispondente a una **espressione regolare** (nel caso più semplice una sottostringa) passata come argomento.

Es. per cercare una stringa di nome "prova" all'interno dell'output di ls:

```
ls | grep prova
```

- Da qui in avanti faremo riferimento alla variante di grep che supporta le espressioni regolari “moderne”: **egrep**



# Espressioni regolari moderne (o estese)

- **egrep** utilizza una sintassi per la specifica dei pattern di ricerca detta espressione regolare (regexp o **RE**). La documentazione è reperibile nella man page `regex(7)`.
- In sintesi
  - **RE** = uno o più **rami** non vuoti separati da |
  - **ramo** = uno o più **pezzi** concatenati tra loro
  - **pezzo** = **atomo** eventualmente seguito da un *moltiplicatore*
  - **atomo** = uno di
    - **(RE)**
    - **[charset]**
    - **^ o \$ o .**
    - **backslash sequence**
    - **Singolo carattere**





# Espressioni regolari moderne (o estese)

## ■ Atomi speciali:

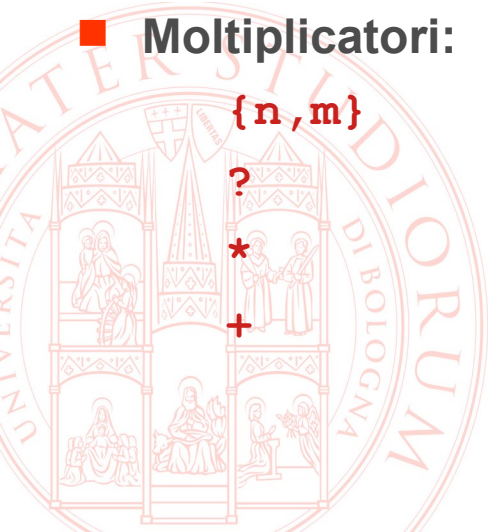
.	indica UN qualsiasi carattere
^	indica l'inizio della linea
\$	indica la fine della linea

## ■ Backslash sequence:

\< – \>	la stringa vuota all'inizio – alla fine di una parola.
\b	la stringa vuota al confine di una parola
\B	la stringa vuota a condizione che non sia al confine di una parola
\w	è sinonimo di “una qualsiasi lettera, numero o _”
\W	è un sinonimo “un qualsiasi carattere non compreso in \w”

## ■ Moltiplicatori:

{n,m}	indica da n a m occorrenze dell'atomo che lo precede
?	indica zero o una occorrenza dell'atomo che lo precede
*	indica zero o più occorrenze dell'atomo che lo precede
+	indica una o più occorrenze dell'atomo che lo precede





# Espressioni regolari moderne (o estese)

## ■ Charset – esempi:

- `[abc]` indica UN qualsiasi carattere fra a, b o c
- `[a-z]` indica UN qualsiasi carattere fra a e z compresi
- `[^dc]` indica UN qualsiasi carattere che non sia né d né c

## ■ Charset basati su character class

`[ :NOME_CLASSE : ]`

dove *NOME\_CLASSE* deve appartenere all'insieme definito in `wctype(3)` come tipicamente:

<code>alnum</code>	<code>digit</code>	<code>punct</code>	<code>alpha</code>	<code>graph</code>	<code>space</code>
<code>blank</code>	<code>lower</code>	<code>upper</code>	<code>cntrl</code>	<code>print</code>	<code>xdigit</code>

o eventualmente nel *locale* attivo

# grep – regole di matching

## ■ *Greediness*

- Nel caso in cui una RE possa corrispondere a più di una sottostringa di una data stringa, la RE corrisponde a quella che inizia per prima nella stringa
- Se a partire da quel punto, la RE può corrispondere a più di una sottostringa, selezionerà la più lunga.


## ■ Priorità nelle RE multilivello

- Anche le sottoespressioni selezioneranno sempre le sottostringhe più lunghe possibili
  - salvo il vincolo che l'intera corrispondenza sia la più lunga possibile
  - dando la priorità alle sottoespressioni che iniziano prima nella RE su quelle che iniziano dopo

Si noti che le sottoespressioni di livello superiore hanno quindi la priorità sulle loro sottoespressioni di componenti di livello inferiore.



# Espressioni regolari - esempi

- Iniziamo osservando che molti caratteri speciali delle RE sono anche caratteri speciali della shell – quando possibile, a scanso di equivoci, è meglio racchiudere l'intera RE tra apici
- `egrep '^Nel.*vita\.$' miofile`  
ha come output tutte le righe di miofile che iniziano per **Nel** e finiscono per **vita.**  

- `egrep '.es[^es]{3,5}e' miofile`  
ha come output tutte le righe che contengono in qualsiasi posizione la sequenza:
  - 1 carattere qualsiasi
  - **es**
  - una sequenza di 3-5 caratteri potenzialmente diversi uno dall'altro a patto che ognuno sia diverso da **e** ed **s**
  - **e**
- Per più esempi, basta andare online, es.  
<https://www.cyberciti.biz/faq/grep-regular-expressions/>
- Ci sono anche tester online ma attenzione al dialetto delle RE usato!

# Grep – opzioni principali

## ■ Controllo del tipo di matching:

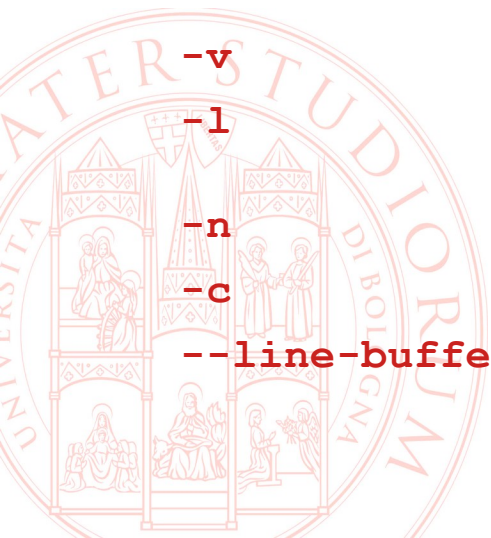
- E** usa le extended RE (come egrep senza parametri)
- F** disattiva le RE e usa il parametro come stringa letterale
- w / -x** fa match solo con RE “whole word” o “whole line”
- i** rende l’espressione insensibile a maiuscole e minuscole

## ■ Controllo dell’input

- r** cerca ricorsivamente in tutti i file di una cartella
- f *FILE*** prende le RE da un **FILE** invece che come parametro

## ■ Controllo dell’output

- o** restituisce solo le sottostringhe che corrispondono alla RE invece della riga che le contiene, separatamente una per riga di output.
- v** restituisce le linee che **non** contengono l’espressione
- l** utile passando a grep più file su cui cercare: restituisce solo i nomi dei file in cui l’espressione è stata trovata
- n** restituisce anche il numero della riga contenente l’espressione
- c** restituisce solo il conteggio delle righe che contengono la RE
- line-buffered** disattiva il buffering



# Modifiche più complesse

- Esistono altri comandi che consentono operazioni complesse sui file, come **sed** e **awk**.
- È riduttivo chiamarli filtri, dispongono di un vero e proprio linguaggio di programmazione per effettuare operazioni di manipolazione del testo.
- Vediamo solo qualche esempio pratico di utilità frequente



# sed

- *Stream Editor*

- Formato base: `sed -e 'comando' o sed -f 'script'`

- Non useremo script generici ma il solo comando di sostituzione

`sed 's/VECCHIO_PATTERN/NUOVO_VALORE/[modificatori]'`

- sostituisce in ogni riga il **NUOVO\_VALORE** alla parte di testo coincidente con **VECCHIO\_PATTERN**

- con `sed -E` i pattern sono *circa* quelli di **egrep**

- Es. inserisce la stringa “Linea:” all'inizio di ogni riga di passwd:

`cat /etc/passwd | sed 's/^/Linea:/'`

- Modificatori del comando di sostituzione

**i**

case insensitive

**g**

global (sostituisce tutte le occorrenze sulla riga)

**NUM**

sostituisce solo l'occorrenza **NUM**-esima

- Opzioni sulla riga di comando

**-i [SUFFIX]**

edita il file dato [backup con estensione SUFFIX se fornita]

**-u**

unbuffered

# tr

- Per sostituire più rapidamente singoli caratteri (senza regex), si può utilizzare **tr** – qualche esempio:

- trasforma ordinatamente le maiuscole in minuscole

```
tr 'A-Z' 'a-z'
```

- sostituisce qualsiasi occorrenza dei caratteri nel primo set con ,

```
tr ' ; : . ! ? ' ' , ' '
```

- in generale, se il secondo set è più limitato del primo set, il suo ultimo carattere viene ripetuto quanto basta a generare la corrispondenza 1:1.

```
tr ' ; : . ! ? ' ' , - ' '
```

In questo caso quindi ; → ,   : → -   . → -   ! → -   ? → -

- elimina ogni occorrenza del carriage return

```
tr -d '\r'
```



# awk

- awk è un interprete per AWK (POSIX 1003.1), un linguaggio Turing-completo, definito data-driven in quanto pensato per applicare un algoritmo a ogni riga di testo fornita in ingresso.
- Noi lo useremo solamente come evoluzione di cut, perché permette di considerare qualsiasi sequenza di caratteri come un unico delimitatore. Nell'uso più comune permette di superare uno dei più evidenti limiti di **cut** in presenza di più delimitatori consecutivi: ad esempio, **cut -f2 -d' '** se ci sono 2 spazi dopo il primo campo considera il 2° spazio come 2° campo.
- Es. stampa il secondo campo del file, purchè sia separato dal primo da un numero qualunque di blanks

```
cat personale | awk '{print $2}'
```

- Es. In un file che riporta il risultato di un'operazione come  
[stringhe varie...] stat=esito [stringhe varie...]  
estrae tutti gli esiti:

```
cat log | awk -F 'stat=' '{print $2}' | awk '{print $1}'
```

- A differenza di cut, non ha il concetto di “-f 5-” (dal quinto campo in poi), ma:

```
cat file | awk '{print substr($0, index($0,$5)) }'
```

- Per approfondire: <http://awklang.org/>



# Variazioni sul tema filtri

- costruire pipeline con comandi che vogliono dati come parametri anziché come stream di input
  - xargs
  - process substitution
  - tee
  - command substitution
- processing di più file di testo
  - diff
  - paste
  - join



# Costruzione di linee di comando con xargs

- Può essere necessario inserire in una pipeline comandi che non leggono stdin, ma vogliono parametri sulla riga di comando
- `xargs <comando>` si aspetta sullo standard input un elenco di stringhe, ed invoca poi comando con tali stringhe come argomenti.
- Es.

`pipeline | cheproduce | nomi_di_file | xargs ls -l`

lancerà `ls -l` per ogni file ricevuto dalla pipeline

- Particolarità del comportamento di xargs
  - xargs raggruppa le invocazioni in modo da ridurre il carico. Questo può funzionare con comandi come `ls`, ma non se il comando accetta un singolo parametro
  - xargs passa le righe dell'input così come sono sulla riga di comando costruita. La presenza di spaziatori quindi farà percepire al comando invocato una molteplicità di parametri

## ■ Opzioni utili:

`-0` (zero)

utilizza null, non lo spazio, come terminatore di argomento

`-L MAX`

usa al più **MAX** linee di input per ogni invocazione

`-p`  
comando

chiede interattivamente conferma del lancio di ogni

# Ridirezioni speciali – process substitution

- Supponiamo di avere `cmd_producer_su_stdout` che in “stile filtro” genera dati su stdout
- Supponiamo che `cmd_consumer_da_file` non accetti stdin, e voglia invece come parametro un file da leggere ed elaborare
- Non si possono connettere con una pipe
- Si potrebbe usare un file temporaneo

```
cmd_producer_su_stdout > tmpfile
```

```
cmd_consumer_da_file tmpfile
```

- Svantaggi?

- Soluzione: *process substitution*

```
cmd_consumer <(cmd_producer_su_stdout)
```

- Il processo tra parentesi è lanciato concorrentemente all'altro e la shell genera un nome di file (sarà una named pipe) da fornire al primo

- Caso simmetrico:

```
cmd_producer_su_file >(cmd_consumer_da_stdin)
```

# tee

- **tee** è un comando utile per duplicare uno stream di output

- invia una copia del proprio stdin a stdout
- invia una copia identica in un file passato come parametro
- opzione **-a**: il file è aperto in append

**comando1 | tee FILE | comando2**

- si noti che combinato alla process substitution permette anche varianti più complesse

```
ls | tee >(grep foo | wc > foo.count) |  
    tee >(grep bar | wc > bar.count) |  
    grep baz | wc > baz.count
```



# shuf

- genera permutazioni random delle righe di stdin o di un file

- esattamente come `sort -r`

- ma anche...

- degli argomenti passati, se attiva l'opzione `-e`

Es. `shuf -e uno due tre`

due

uno

tre

- di un range di numeri, con l'opzione `-i`

Es. `shuf -i 1-5`

5

1

3

2

4

- ripetendo all'infinito estrazioni casuali degli argomenti, con l'opzione `-r`

# Generazione di parametri – command substitution

- La *command substitution* permette di utilizzare direttamente un processo per generare parametri da collocare sulla command line per un altro comando

- Due sintassi

``comando``

`$(comando)`

- equivalenti a meno di piccoli dettagli sul trattamento di alcuni caratteri speciali che potrebbero apparire in **comando**
- **comando** viene eseguito in una subshell
- il suo stdout compare sulla riga di comando al posto del token di command substitution

- Es.:

`ls $(cat /etc/passwd | cut -f6 -d:)`

- estrae le home dir degli utenti e le pone come parametri a **ls**

# Strumenti che operano su più file – diff

- Più frequentemente usato in modo interattivo, ma utile anche negli script, diff permette di mostrare le differenze tra due file.

```
diff filesinistro filedestro
```

```
2d1
```

```
< riga due
```

deleted

```
4c3
```

```
< riga quattro
```

changed

```
---
```

```
> riga 4
```

```
5a5
```

```
> riga 5bis
```

added

```
filesinistro
```

```
filedestro
```

```
riga uno
```

```
riga uno
```

```
riga due
```

```
riga tre
```

```
riga tre
```

```
riga 4
```

```
riga quattro
```

```
riga cinque
```

```
riga cinque
```

```
riga 5bis
```

# Strumenti che operano su più file – paste

- In un certo senso l'opposto di **cut**: unisce “orizzontalmente” le righe di posizione omologa in vari file

**paste** **nazioni** **superfici** **abitanti**

Italia 301230 61261000

Francia 547030 65630000

Germania 357021 81305000

Olanda 41543 6730000

**nazioni**

Italia

Francia

Germania

Olanda

**superfici**

301230

547030

357021

41543

**abitanti**

61261000

65630000

81305000

6730000



# Strumenti che operano su più file – join

- Stesso principio di **paste**, ma anziché selezionare le righe sulla base della posizione, le unisce se iniziano con la stessa “chiave” (necessita di file ordinati in modo identico sulla chiave selezionata)

```
join db_superfici db_abitanti
```

```
Italia 301230 61261000
```

```
Francia 547030 65630000
```

```
Germania 357021 81305000
```

```
Olanda 41543 6730000
```

**db\_superfici**

Italia 301230

Francia 547030

Germania 357021

Olanda 41543

**db\_abitanti**

Italia 61261000

Francia 65630000

Germania 81305000

Olanda 6730000

